View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Scholarly Materials And Research @ Georgia Tech

Write-Optimized Indexing for Log-Structured Key-Value Stores

Yuzhe Tang [†] Arun Iyengar [‡] Wei Tan [‡] Liana Fong [‡] Ling Liu [†]

[†]*Georgia Institutue of Technology, Atlanta, GA, USA , Email:* {yztang@, lingliu@cc.}gatech.edu
[‡]*IBM T.J.Watson Research Center, Yorktown, NY, USA , Email:* {aruni, wtan, llfong}@us.ibm.com

Abstract

The recent shift towards write-intensive workload on big data (e.g., financial trading, social user-generated data streams) has pushed the proliferation of the log-structured key-value stores, represented by Google's BigTable, HBase and Cassandra; these systems optimize write performance by adopting a log-structured merge design. While providing key-based access methods based on a Put/Get *interface, these key-value stores do not support value-based access methods, which significantly limits their applicability in many web and Internet applications, such as real-time search for all tweets or blogs containing "government shutdown". In this paper, we present* HINDEX*, a write-optimized indexing scheme on the log-structured key-value stores. To index intensively updated big data in real time, the index maintenance is made lightweight by a design tailored to the unique characteristic of the underlying log-structured key-value stores. Concretely,* HINDEX *performs* append-only *index updates, which avoids the reading of historic data versions, an expensive operation in the log-structure store. To fix the potentially obsolete index entries,* HINDEX *proposes an offline index repair process through tight coupling with the routine compactions.* HINDEX*'s system design is generic to the* Put/Get *interface; we implemented a prototype of* HINDEX *based on HBase without internal code modification. Our experiments show that the* HINDEX *offers significant performance advantage for the write-intensive index maintenance.*

I. Introduction

In the big data era, the key-value data updated by intensive write streams is increasingly common in various application domains, such as high-frequency financial trading, social web applications and large network monitoring. To manage the write-intensive workload, various log-structured key-value stores, abbreviated as LSKV store, recently emerged and become prevalent in the real-world production use; this list includes Google's BigTable [1], Facebook's Cassandra [2], [3], Apache's HBase [4] and among many others. These key-value stores are based on a log-structured merge design [5], which optimizes the write performance by append-only operations and lends themselves to the write-intensive workloads.

The existing key-value stores typically expose key-based data access methods, such as Put and Get based on the primary data keys. While such interface works for basic workloads, the lack of a value-based access method limits the applicability of an LSKV store in the modern web and Internet applications. For example, a simple search based on the value attribute such as "finding all tweets or blogs containing 'government shutdown' " would lead to a full-table scan in the store, resulting in an unacceptable latency for end users.

To enable value-based access methods, we propose HINDEX to support secondary indexing on generic LSKV stores. The HINDEX is an indexing middleware that sits on top of generic LSKV stores; Together with the Put/Get interface, it provides a unified framework to support both key-based and value-based data access to key-value stores. The system design of HINDEX is optimized towards a write-intensive workload on LSKV stores. In this setting, there are two features that are desirable: 1) *Real-time data availability*: In the presence of update-intensive data, index should be updated in real time to reflect the latest version of an (evolving) object to the end users. In many real-world scenarios, the end users are mostly interested in the latest data, such as the latest score in a game, the latest online news, or the latest bids in a web auction. 2) *Write-optimized index maintenance*: In order to catch up with intensive write stream, the index maintenance has to be optimized on writes. However, it is challenging to maintain a real-time index in a write-optimized manner; because these two goals favor essentially opposite index designs. Upon a data update, the real-time indexing needs to apply the update to the index structure as early as possible, while the write-optimized indexing may prefer delay or even avoid the action of updating the index so that the extra write cost associated with the action can be saved. To strike a balance in the trade-off between the two ends, we propose the use of append-only indexing design for HINDEX, based on the unique performance characteristic

of underlying LSKV stores. Concretely, upon data updates, HINDEX performs the append-only Put operations to the index structure; it deliberately avoids the deletion of obsolete versions and thus avoids the reading of historic data using Get operations. The rationale behind this design is based on the fast-write-slow-read performance of LSKV stores; in a typical setting, the latency of a Get operation in an LSKV store can be significantly higher than that of a Put (e.g. by an order of magnitude). The detailed explanation will be given in Section II. While it optimizes the index update costs, the append-only design may cause an eventual inconsistency between the index structure and the base data. To fix the index inconsistency, an index repair process is used that conceptually runs a batch of Get's to find the obsolete index entries and then deletes them. We propose a novel design that *defers the index repair to the offline compaction time*. Here, a compaction is a native maintenance routine in the LSKV stores. Coupling the index repair with the compaction can save the repairing overhead substantially. This is based on our key observation that the Get performance in an LSKV store is significantly affected by the compaction process. To verify this observation, we conducted performance study on the most recent HBase release (i.e., HBase 0.94.2). A preview of experiment results is shown in Figure 1; the Get latency after compaction is much faster (with more than 7× speedup) than the latency before.[1]

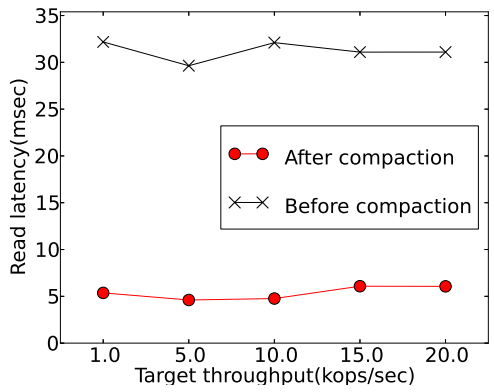


Fig. 1: Read latency before/after compaction

In summary, our contributions in this work are following:

- We have coined the abstraction of LSKV stores for various industrial strength big-data storage systems and propose HINDEX to extend this abstraction with real-time value-based access methods.
- We optimize the HINDEX performance towards write-intensive workload by taking into account of the unique performance characteristic of underlying LSKV stores. By carefully deferring certain part of index maintenance, it makes the real-time indexing lightweight and unintrusive to the system online performance. The deferred computations (i.e. index repair) are made efficient as well, by coupling with the native compaction routine.
- The HINDEX system is generic and adaptable. It can be integrated to any LSKV store and relies solely on the generic Put/Get interface. To demonstrate the ease of implementation, we have built a complete prototype of HINDEX on HBase, without any code change on the HBase side. With this prototype, our real-world experiments show that HINDEX can offer significant performance improvement on write-intensive workload comparing to the status quo indexing approaches.

[1]For details, please refer to the experiments in Section VI-B and the explanation in Section II.

II. Background: LSKV stores

In this section, we present a background introduction on the LSKV stores. Due to the proliferation of write-intensive workloads, many emerging key-value stores fall under the category of a LSKV store, including Google's BigTable [1]/LevelDB [6], Apache's HBase [4] and Cassandra [3], and recently proposed RocksDB [7] by Facebook. These scalable stores expose a *key-value data model* to client applications and internally adopt the design of log-structured merge tree (or LSM tree) that optimizes the write performance.

In a key-value data model, a data object is stored as a series of key-value records – each object is identified by a unique key k and associated with multiple overwriting versions. Each version has a value v and a unique timestamp ts. To retrieve and update an object, LSKV store exposes a simple Put/Get interface: Put(k,v,ts), Delete(k,ts) and Get$(k) \rightarrow \{< v,ts >\}$. Note that the Get operation only allows for key-based access.

LSM Tree-based Data Persistence

LSKV stores adopt the design of LSM tree [5], [8] for its local data persistence. The core idea is to apply random data updates in append-only fashion, so that most random disk access can be translated to efficient, sequential disk writes. The read and write paths of an LSM tree are shown in Figure 2. After the LSM tree locally receives a Put request, the data is first buffered[2] in an in-memory area called *Memstore*[3], and at a later time is flushed to disk. This Flush process sorts all data in Memstore based on key, builds a key-based cluster index (called block index) in batch, and then persists both the sorted data and index to disk. Each Flush process generates an immutable file on disk, called *HFile*. As time goes by, multiple Flush executions can accumulate multiple HFiles on disk. On the data read path, an LSM tree process a Get request by sifting through existing HFiles on disk to retrieve multiple versions of the requested object. Even in the presence of existing in-memory index schemes (e.g., Bloom filters and the block index used in HBase), an LSM tree still has to access multiple files for a Get, because

[2]For durability, LSM trees often have option for write-ahead logging (or WAL) before each write to buffer.

[3]In this paper, we use the HBase terminology to describe an LSM tree.

the append-only writes put multiple versions of the same object to different HFiles in a non-deterministic way. This design renders LSKV store to be a write-optimized system since a data write causes mostly in-memory operations, while a read has to randomly access the disk, causing disk seeks.

A LSKV store exposes a Compact interface for system administrator to perform the periodical maintenance routine, usually in offline hours. A Compact call triggers the compaction process, which merges multiple on-disk HFiles to one and performs data cleaning jobs to reclaim disk space from obsolete object versions. The Compact consolidate the resource utilization in the LSKV store for further operation efficiency.

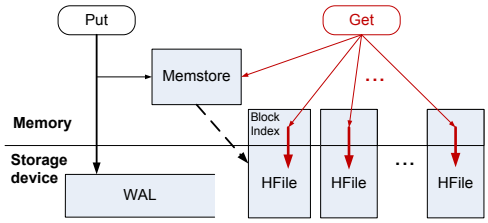


Fig. 2: System architecture of an LSM tree: The figure shows the data write and read path resp. by black and red arrows. The thick arrows represent disk access.

III. The HINDEX Structure

In this section we present the system and data model used in HINDEX, and then describe the index materialization in an underlying LSKV store.

A. System and Data Model

The overall system where HINDEX is positioned is a cloud serving data center, in which the server cluster is organized into a multi-tier architecture. The application tier prepares data for writes or for query processing, and the storage tier is responsible for persisting data. The LSKV store resides in the storage tier. In this architecture, as shown by Figure 3, HINDEX is a middleware that resides between the application tier and storage tier: It exposes programming interfaces to the application clients, while it changes certain behavior of the LSKV store. In this paper, we refer to the application server the "client", since it is the client to the key-value store. Note this client still resides inside Cloud serving data center. Internally, HINDEX rewrites the application-level function calls into Put/Get operations of the LSKV store.

The extended key-value model: The data model exposed by HINDEX is an extended key-value model in the sense that in addition to the key-based access, HINDEX adds a value-based access method. Given multi-version object, we adopt a general *m-versioning* policy, which considers valid and fresh the latest $m > 1$ versions instead of one. HINDEX has the following API:

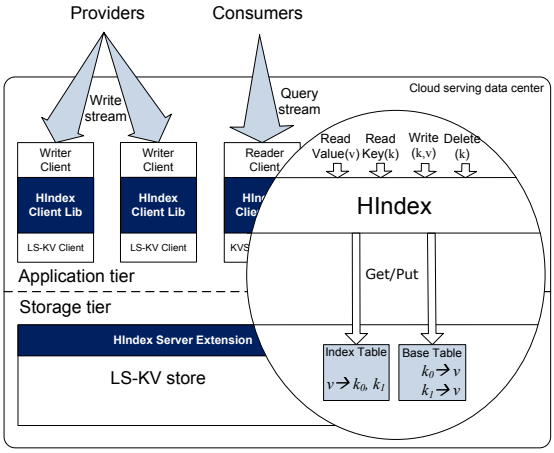


Fig. 3: HINDEX architecture

- Write(k,v,ts): Given a row key k, it updates (or inserts) the value to be v with timestamp ts.
- Remove(k,ts): Given a row key k and a timestamp ts, it deletes all previous versions with timestamp before ts.
- ReadKey$(k,ts,m) \rightarrow \{< k,v',ts' >\}_m$: Given a row key k, it returns the value versions before ts, that is, $ts' \leq ts$. Given a versioning number m, the method returns the latest m versions of the requested key.
- ReadValue$(v,ts,m) \rightarrow \{< k',v,ts' >\}$: Given value v in an indexed column, it retrieves all the row keys k' whose values are v and which are valid under the m-versioning policy. That is, the result version ts' must be among the latest m versions of its key k' as of time ts.

The first three methods are similar to the existing key-based Put/Get interface, while the last one is for value-based access. In the API design, we expose timestamp ts which allows the client applications to fully specify the consistency requirement. In practice, generating a unique timestamp, if necessary, can be done with the help of the existing timestamp oracles [9], [10]. HINDEX, being a general index maintenance protocol, can be adaptable to different index and query types. Throughput this paper, we mainly use the exact-match query and indexing as the use case, although the HINDEX paradigm can be easily extended to other query types; For instance, the keyword query can be supported by parsing the keywords from value v when maintaining or updating the index.

B. Index Materialization

To support global indexing in HINDEX, the index structure is materialized as a hidden table inside the underlying LSKV store. In terms of structure, this index table is nothing special to a regular sharded table in the LSKV store, except it is hidden and invisible from the client applications' perspective. The index table is fully managed by our HINDEX library. The data inside the index table is an inverted version of the base data; As shown in Figure 3, the index entries are keyed by the value of the original key-value data. For different keys

associated with the same value in the base table, HINDEX materializes them in the same row in the index table but as different versions.

IV. Online HINDEX: Maintenance and Query Evaluation

This section describes the online operations in HINDEX which includes the index maintenance and value-based query evaluation.

A. The Index Maintenance

To motivate the append-only design of index maintenance in HINDEX, we first look at a baseline approach based on the traditional update-in-place technique. The update-in-place index is widely used in today's database systems and is applied on recent scalable indexes in cloud [11], [12].

TABLE I: Algorithms for online writes and reads

Algorithm 1 Write(key k, value v, timestamp ts)

```
1: index.Put(v,k,ts)
2: base.Put(k,v,ts)
```

Algorithm 2 ReadValue(value v, timestamp ts, versioning m)

```
1: {<k,ts'>} ← index.Get(v,ts)                          ▷ ts' ≤ ts
2: for ∀ <k,ts'> ∈ {<k,ts'>} do
3:     {<k,v',ts''>} ← ReadKey(k, ts, m)
4:     if ts' ∈ {ts''} then
5:         result_list.add({<k,v,ts'>})
6:     end if
7: end for
8: return result_list
```

An update-in-place baseline: The update-in-place approach causes two index-updating actions in the data write path. That is, upon a Write(k,v,ts) call, in addition to updating the base table, the approach issues 1) a Put of new index entry $<v,k,ts>$ to the index table and 2) a Get to the base table that reads the latest versions of the updated key k, based on which it determines if there is any version obsoleted because of this update. If a version, say $<k,v_0,ts_0>$, is found to be obsoleted, it further issues a Delete to the index table that deletes the newly obsoleted index entry, $<v_0,k,ts_0>$. We call the first action an index append and the second action an index repair. Note that the index append action causes Put-only operations and the index repair action incurs expensive Get operation for each data update. The update-in-place design, by synchronously executing both actions, leads to significant amplification of per-write cost (due to the expensive Get), thus considerably decreasing the write throughput.

In HINDEX, we made the design choice to execute only the Put operations synchronously with data updates and and defer the expensive index repair action. By this way, the online write simply appends the new entry to the index table at very low cost, which achieves real-time queriability yet without sacrificing the write throughput. Algorithm 1 shows the online Write algorithm. Note that the two Put calls are specified with the same timestamp ts.

B. Online Read Query Evaluation

Given the append-only indexing design, the query evaluation of ReadValue checks both the index and base tables for the fresh result. It is necessary to check the base table because of the potential existence of obsolete entries in the index table, caused by the append-only index maintenance. In addition, the obsolete index entries can only be discovered by checking the base table where a full history of versions of a data object are maintained in the same place. Algorithm 2 illustrates the evaluation of query ReadValue(v,ts,m): It first reads all the index entries of the requested value v before timestamp ts. This is done by a Get operation in the index table. For each returned index entry, say ts', it needs to be determined whether the entry is obsolete in an m-versioning sense. To do so, the algorithm reads the base table by issuing a ReadKey query (which a simple wrapper of a Get call to the base table), which returns all the latest m versions $\{ts''\}$ before timestamp ts. Depending on whether ts' show up in the list of $\{ts''\}$, the algorithm can then decide if it is obsolete. Only when the version is not obsolete, it is then added to the final result.

C. Implementation

Online HINDEX can be implemented on LSKV stores by two approaches. The first approach is to implement it in the client library. That is, the client directly coordinates all the function calls (e.g. Write and ReadKey) as in the Write and ReadValue algorithms; This is possible since all the operations in these algorithms are based on the generic Put/Get interface. The second approach is server-side implementation. In this case, the index and base table servers play the role of coordinators to execute the read and write algorithms. In particular, the Write is rewritten to a base-table Put by the HINDEX client library. When the base-table Put gets executed in the base table, it also triggers the execution of the index-table Put. Likewise, the ReadValue is rewritten to an index-table Get call, upon the completion of which the index table triggers the execution of the base-table Get, if needed. The server-side implementation favors the case where application servers and storage servers are located in different clusters and the cross-boundary inter-cluster communications are more expensive than the intra-cluster communications. The server-side implementation can be done by directly modifying the code of an LSKV store system, or as in our implemented prototype, by adding server extensions based on the extension interface of LSKV stores (described in Appendix A).

V. Offline HINDEX: Batched Index Repair

In HINDEX, the index repair process eliminates the obsolete index entries and can keep the index fresh and up-to-date. This section describes the system design and implementation of offline HINDEX operations for the batched index repair.

A. Computation Model and Algorithm

To repair the index table, it is essential to find obsolete data versions. A data version, say $< v_1, k, ts_1 >$, is considered to be obsolete when either of the following two conditions is met.

1. There are at least m newer key-value versions of key k that exist in the system.
2. There is at least one newer Delete tombstone[4] of key k that exists in the system.

To find all the obsolete versions or data garbage currently present in the system, we start from the base table; Because the base table has the data records sorted in the key order, which helps verify the above two conditions. To be specific, we scan the base table while performing garbage collection. Algorithm 3 illustrates the batched garbage collection algorithm on a data stream coming out of the table scan. Basically, it assumes the table scan throws a key-value data stream ordered by key and then timestamp. The algorithm maintains a queue of capability m and emit the version for the index deletion only when the version has gone through the queue (meaning it's older than at least m versions which are still in the queue) and it is not too older. This extra condition considers the case of a very old version that might has already been repaired in the last round of offline compaction (i.e. with timestamp ts before the last compaction time ts_{Last}). In the algorithm, it also considers the condition regarding a Delete tombstone; It will emit all the versions before the Delete tombstone marker. Note that our algorithm runs in one pass and maintains a small memory footprint (i.e. the m-sized queue).

[4]In an LSKV store, a Delete operation appends a tombstone marker in the store without physically deleting the data.

Algorithm 3 BatchedGarbageCollection(Key-value stream s)

```
1: for ∀Key-value data kv ∈ s do        ▷ Stream sorted by key and time (in descending order)
2:     if k_Current == kv.k then
3:         if queue.size() < m then
4:             queue.enqueueToHead(kv)
5:         else if queue.size == m then
6:             queue.enqueueToHead(kv)
7:             kv' ← queue.dequeueFromTail()
8:             if kv'.ts ≥ ts_Last then        ▷ kv.ts is no older than ts_Last
9:                 emit(kv')
10:            end if
11:        end if
12:    else
13:        loop queue.size()> 0        ▷ Clear the queue for the last key
14:            kv' ← queue.dequeueFromHead()
15:            if kv' is a Delete tombstone then
16:                emit(queue.dequeueAll())
17:            end if
18:        end loop
19:        k_Current ← kv.k
20:        queue.enqueueToHead(kv)
21:    end if
22: end for
```

B. A Compaction-aware System Design

The index repair entails a table scan for collecting obsolete versions. To materialize the table scan in the presence of offline compaction process, one can have three design options, that is, to run the index repair 1) before the compaction, 2) after the compaction or 3) coupled inside the compaction. In HINDEX, we made the choice to adopt the last two options to either couple the index repair with the compaction or after the compaction. Recall that in an LSKV store, the read performance is significantly improved after the compaction. The rationale is that table scan, being a batch of reads, also has its performance dependent on the execution of the compaction and the number of HFiles; Without compaction, there would be a number of HFiles and a key-ordered scan would essentially become a batch of random reads that make the disk heads swing between the on-disk HFiles.

The offline HINDEX runs in three stages; As illustrated in Figure 4, it runs offline compaction, garbage collection and index garbage deletion. After a Compact call is issued, the system would run the compaction routine, which also triggers the execution of batched index repair. In the index repair process, the garbage collection identifies the obsolete data versions and emit them to the next-stage garbage deletion. The index garbage deletion issues a batch of deletion requests to the distributed index table. In the follows, we describe our subsystem design for each stage and discuss the design options.

1) The Garbage Collection: We present two system designs for garbage collection, including an isolated design that puts the garbage collection right after the compaction process, and a pipelined design that couples the garbage collection inside the compaction process.

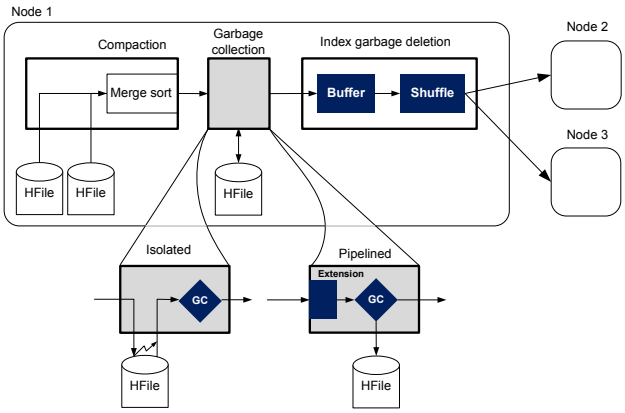


Fig. 4: Compaction-aware system design for offline batched index repair

An isolated design: The garbage collection subsystem is materialized as an isolated component that runs after the previous compaction completes. As portrayed in Figure 4, the system monitors the local file store and the number of HFiles in it. When an offline compaction process finishes, it reduces the number of HFiles to one, upon which the monitor component triggers the garbage collection process. In this case, the garbage collection reloads the newly generated HFile to memory (at the moment the file system cache may very likely be hot), scan through it, and run Algorithm 3 to collect the obsolete data versions. This system design is generic since it does not rely on anything internal of an LSKV store.

A pipelined design: Alternatively, the garbage collection subsystem can be implemented by pipelining the compaction's output stream directly to the garbage collection service. To be specific, as shown in Figure 4, the pipelined garbage collection intercepts the output stream from compaction while the data is still in memory; The realization of interception is described in the next paragraph. Then it runs the garbage collection computation in Algorithm 3; If the data versions are found not to be obsolete, they are persisted to the newly merged HFile, otherwise they are emitted without persistence. By this way, we can guarantee that the new HFile does not contain the any data versions that are already repaired (In this case, Line 8 in Algorithm 3 will always be true.) Comparing the isolated design, the pipelined design saves disk accesses, since the data stream is pipelined in memory without being reloaded from disk.

Implementation note: Interception of the compaction's output stream can be realized by multiple ways. The most straightforward and generalized way is to directly modify the internal code of an LSKV store. Another way is to rely on the extension interface widely available in existing LSKV stores (described in Appendix A) which allows for an add-on and easier implementation. In particular, our HBase-based prototype implements the pipelined design using extension-based implementation; We register a CoProcessor callback function to hook the garbage collection code inside the compaction. By this way, our implementation requires no internal code change of HBase, and can even be deployed lively onto a running HBase cluster.

2) The Index Garbage Deletion: For each the key-value record emitted from the garbage collection, it enters the garbage deletion stage; The record is first buffered in memory and later shuffled before being sent out by a Delete call to the remote index table. The shuffle process sorts and clusters the data records based on the value. By this way, the reversed value-key records with the same destination can be packed into a single serializable object in a RPC call, thus network utilization can be saved. In design the garbage deletion subsystem, we expose a tunable knob to configure the maximal buffer size and adapt to system resource utilization; The bigger the buffer is, the less bandwidth overhead it can achieve at the expense of more memory overhead.

VI. Experiments

This section describes our experimental evaluation of HINDEX. We first did experiments to study the performance characteristics of HBase, a representative LSKV store, and then to study HINDEX's performance under various micro-benchmarks and a synthetic benchmark with comparison to alternate design approaches and architectures. Before all this, we describe our experiment system and platform setup.

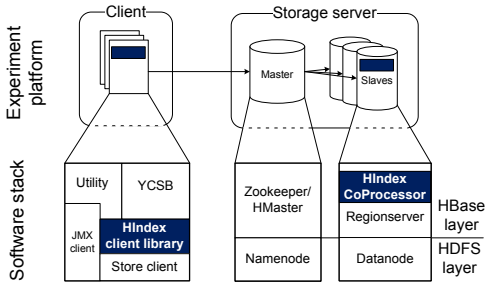


Fig. 5: Experiment platform and HINDEX deployment

A. Experiment Setup

The experiment system, as illustrated in Figure 5, is organized in a client/server architecture. In the experiment, we used one client node and a 19-node server cluster, consisting of a master and 18 slaves. The client connects to both the master and the slaves. We setup the experiment system by using Emulab [13], [14]; All the experiment nodes in Emulab are homogeneous in the sense that each machine is equipped with the same 2.4 GHz 64-bit Quad Core Xeon processor and a 12 GB RAM. In terms of the software stack, the server cluster used both HBase and Hadoop's HDFS [15]. The HBase and HDFS clusters are co-hosted on the same set of nodes. As shown in Figure 5, the master node serves a Zookeeper instance and an HMaster instance at the HBase layer, and a namenode instance at the HDFS

layer. Each of the 18 slave nodes co-hosts a region server for HBase and a data node for HDFS. Unless otherwise specified, we used the default configuration in the out-of-box HBase for our performance study. The client side is based on YCSB framework [16], an industry-standard benchmark tool for key-value stores. The original YCSB framework generates only key-based queries, and for the purpose of our experiment, we extended the YCSB to generate value-based queries. We use the modified YCSB framework to drive workload into the server cluster and measure the query performance. In addition, we collect the system profiling metrics (e.g. number of disk reads) through a JMX (Java management extension) client. For each experiment, we clean the local file system cache.

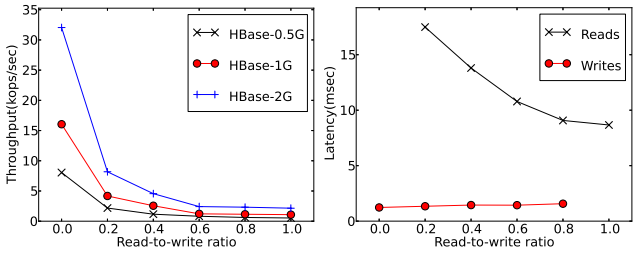
HINDEX *prototype deployment:* We have implemented an HINDEX prototype in Java and on top of HBase 0.94.2. The HINDEX prototype is deployed to our experiment platform in two components; as shown by dark rectangular in Figure 5, it has a client-side library and a server-side component connected to HBase's region servers through the CoProcessor interface. In particular, the prototype implements both the isolated garbage collection and pipelined garbage collection in the server component. Our prototype implementation of HINDEX has been put in production development inside IBM.

Dataset: Our raw dataset consists of $1,000,000,000$ key-value records, generated by YCSB using its default parameters that simulates the production use of key-value stores inside Yahoo!. In this dataset, data keys are generated in a Zipf distribution and are potentially duplicated, resulting in $20,635,449$ distinct keys. The data values are indexed. The raw dataset is materialized to a set of data files, which are preloaded to the system in each experiment. For queries, we use $1,000,000$ key-value queries, be it either Write, ReadValue or ReadKey. The query keys are randomly chosen from the same raw dataset, either from the data keys or the data values.

B. Performance Study of HBase

Read-write performance: This set of experiments evaluates the read-write performance in the out-of-box HBase (with default HBase configuration) to verify that HBase is aptly used in a write-intensive workload. In the experiment, we set the target throughput high enough to saturate the system. We configure the JVM (on which the HBase runs) with different heap sizes or memory sizes. We varied the read-to-write ratio [5] in the workload, and report the maximal sustained throughput in Figure 6a, as well as the latency in Figures 6b. In Figure 6a, as the workload becomes more read intensive, the maximal sustained throughput of HBase decreases, exponentially. For different JVM memory sizes, HBase exhibits the similar behavior. This result shows that HBase is not omnipotent but particularly optimized for write-intensive workloads. Figure 6b depicts the latency respectively for reads and writes (i.e. Get and Put) in HBase. It can be seen that the reads are much slower than writes, in an order of magnitudes. This result matches the system model of LSKV store in which reads need to check more than one place for multiple data versions and the writes are append-only and fast. In the figure, as the workload becomes more read intensive, the read latency decreases. Because with read-intensive workload, there are fewer writes and thus fewer data versions in the system for a read to check, resulting in faster read performance.

[5]In the paper, the read-to-write ratio refers to the percentage of reads in a read-write workload.



(a) Maximal sustained throughput (b) Latency

Fig. 6: HBase performance under different read ratios

Read performance and HFiles: This experiment evaluates HBase's read performance under varying number of HFiles. In the experiment, we start with preloading the dataset into the HBase cluster, which results in averagely 11 HFiles in each region sever as shown in Figure 1. During the experiment, we issued every 5 million Put's to the HBase cluster and measure the read latency by issuing a number of Get's then. In this process, we have configured the HBase to disable its automatic compaction and region split and during the read performance evaluation, we disallow any Put operations, so that the number of HFiles would stay constant at that time. We measure the number of HFiles between each Put stage. As shown in Figure 1, the average number of HFiles increases from 11 to 27.5 in the experiment. In the end, we manually issued an offline Compact call across all the regions in the cluster, which should leave all the regions with a single HFile. Then the read latency is measured again. We report the changes of read latency in this process in Figure 7 with the red line. As can be seen, the line of read latency basically matches with that of number of HFiles; As there are more HFiles present in HBase, the read latency also becomes bigger. After the compaction which merge all HFiles into one, the read latency also drops greatly. The experiment shows the strong dependency between the Get latency and the number of HFiles in HBase. In particular, for the state with 27.5 HFiles and the final state with 1 HFiles, we have shown the latency difference previously in Figure 1.

C. HINDEX Performance

Online write performance: This experiment evaluates HINDEX performance under the write-only workloads. We

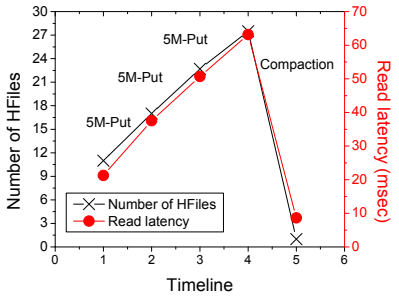
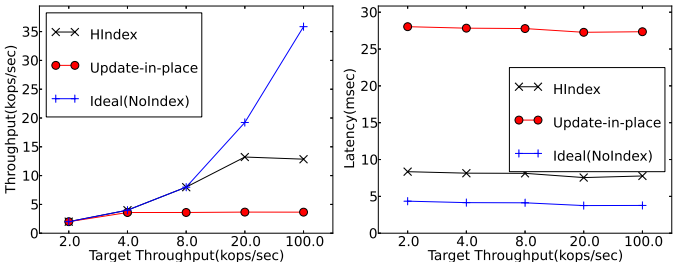


Fig. 7: Read latency with varying number of HFiles

drive the data writes the HBase cluster deployed with our HINDEX prototype. In the experiment, we compare HINDEX with the update-in-place indexing approach described as in Section IV-A. We also consider the ideal case where there is no index structure maintained. The performance results in terms of sustained throughput are reported in Figure 8. As the target throughput increases, the update-in-place indexing approach hits the bottleneck (or saturation point) much earlier than HINDEX. While HINDEX can achieve a maximal throughput at about 14 thousand operations (kops) per second, the update-in-place indexing approach can only sustain at most 4 kops per second. Note that the ideal case can achieve higher throughput but can not deliver the utility of serving value-based queries. This result leads to a 3 $\times$ performance speedup of HINDEX. In terms of the latency, Figure 8b illustrates that HINDEX constantly outperforms the update-in-place approach under scenarios of different throughput.



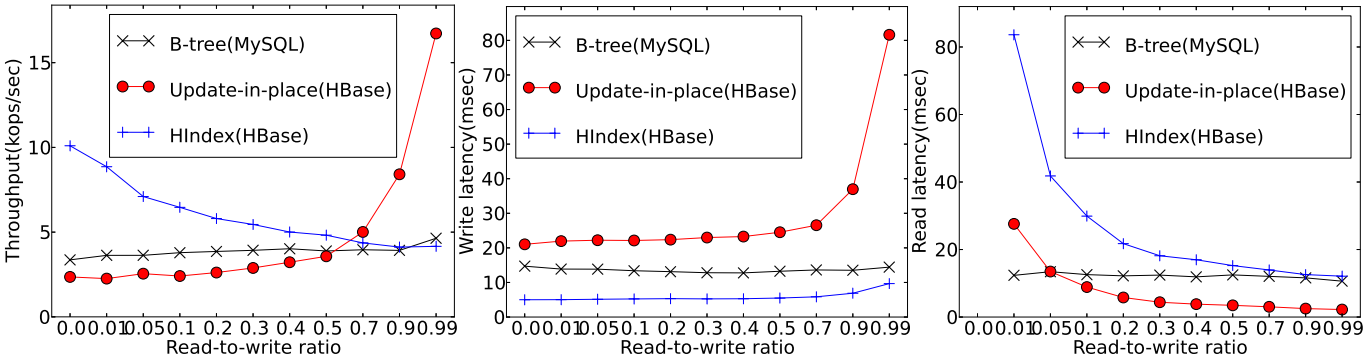
(a) Maximal throughput (b) Latency

Fig. 8: Index write performance

Online read-write performance: In this experiment, we evaluate HINDEX's performance in the workload that varies from read-intensive workloads to write-intensive ones. We compare the HINDEX on top of HBase against two alternative architectures: the B-tree index in MySQL and the update-in-place indexing on HBase. For fair comparison, we use the same dataset in both HBase and MySQL, and drive the same workload there. MySQL is accessible to YCSB through a JDBC driver implemented by us, in which we reduce as much as possible the overhead spent in the JDBC layer. The results are shown in Figure 9. With varying read-to-write ratios, HINDEX on HBase is clearly optimized toward write-intensive workload, as can be seen in Figure 9a. On a typical write-intensive setting with 0.1 read-to-write ratio, HINDEX on HBase outperforms the update-in-place index on HBase by a 2.5$\times$ or more speedup, and the BTree index in MySQL by 10$\times$. When the workload becomes more read-intensive, HINDEX may become less advantageous. By contrast, the update-in-place approach is more read-optimized and the BTree index in MySQL has a stable and relatively inefficient performance, regardless of different workloads. This may be due to that MySQL has made intensive use of locking for full transaction support, an overkill to our targeted use case and workloads. In terms of latency, the HINDEX on HBase has the lowest write latency but at expenses of relatively high read latency due to the needs to read the base table. By contrast, the update-in-place index has the highest write latency due to the need to read obsolete version in HBase, and a low read latency due to that it only reads the index table. Note that in our experiments, we use more write-intensive values for read-to-write ratios (e.g. more ticks in interval $[0, 0.5)$ than in $[0.5, 1.0]$).

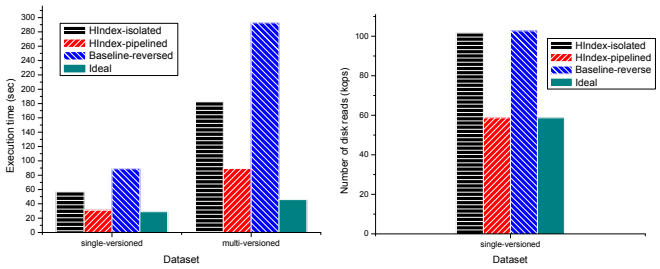
Offline index repair performance: This experiment evaluates the performance of offline index repair with compaction. We mainly focus on the approach of compaction-triggered repair in the offline HINDEX; in the experiment we tested two implementations, with isolated garbage collection and pipelined garbage collection. For comparison, we consider a baseline approach that reverses the design of offline HINDEX, that is, to run the batch index repair before (rather than after) the compaction. We also test the ideal case in which an offline compaction runs without any repair operations. During the experiment, we tested two datasets: a single-versioned dataset that is populated with only data insertions so that each key-value record has one version, and a multi-versioned dataset populated by both data insertions and updates which results in averagely 3 versions for each record. While the multi-versioned data is used to evaluate both garbage collection and deletion during the index repair, the single-versioned dataset is mainly used to evaluate the garbage collection, since there are no obsoleted versions to delete. In the experiment, we have configured the buffer size to be big enough to accommodate all obsolete data in memory. We issued an offline Compact call in each experiment, which automatically triggers the batch index repair process. Till the end, we collect the system profiling information. In particular, we collect two metrics, the execution time and the total number of disk block reads. Both metrics are emitted by the HBase's native profiling subsystem, and we implemented a JMX client to capture the emitted values.

We run the experiment three times, and report the average results in Figure 10. In terms of execution time, we have the results shown in Figure 10a. In general the execution time with multi-versioned dataset is much longer than that with single-versioned dataset, because of the extra need for the index deletion. Among the four approaches, the baseline is the most costly because it loads the data twice and from the



(a) Maximal throughput (b) Write latency (c) Read latency

Fig. 9: Performance comparison between HINDEX in HBase and MySQL



(a) Execution time (b) Number of disk reads

Fig. 10: Performance of offline index repair

not-yet-merged small HFiles, implying most disk reads are random access. The ideal case incurs the least execution time. Between the two HINDEX designs, the pipelined garbage collection requires less execution time because it only needs to load the on-disk data once. To understand the performance difference, it is interesting to look at the disk read numbers, as shown in Figure 10b. We only show the results with the single-versioned dataset, because disk reads only occur in the garbage collection. The baseline approach incurs similar number of disk reads to the HINDEX with isolated design, because both approaches load the data twice from the disk. Note that the disk reads in the baseline approach are most random access while at least half of disk access in the isolated HINDEX should be sequential; this difference leads to differences in their execution time. In Figure 10b, the ideal case has similar cost to the HINDEX with the pipelined design. Because both approaches load on-disk data once. From the single-versioned results in Figure 10a, it can be seen that their execution time is also very close to each other, due to that the extra garbage collection caused by the HINDEX approach is very lightweight and incurs few in-memory computations.

Mixed online and offline operations: In this experiment, we compare HINDEX and the update-in-place indexing approach as a whole package of solution. In other words, we consider the online and offline operations together. Because the update-in-place approach already repairs the index in the online phase, there is no need to perform index repair in the offline time. For fair comparison, we run the offline compaction (without any repair actions) for the update-in-place index. In the experiment, the online workload contains a series of writes and the offline workload simply issues a Compact call and if any, the batch index repair. For simplicity, we here only report the results of pipelined HINDEX. We report the execution time and the disk read number. The results are presented in Table II. In general, HINDEX incurs much shorter execution time and fewer disk reads than the update-in-place approach. For example, the execution time of HINDEX (as bold text in the table) is one third of that of the update-in-place approach. We breaks down the results to the online costs and offline costs, as in the bottom half of the table, which more clearly shows the advantage of having the index repair deferred to the offline phase (recall this is the core design of HINDEX). Although the update-in-place index wins slightly in terms of the offline compaction (see the bold text "279.179" compared to "459.326" in the table), HINDEX wins big in the online computation (see bold text "1093.832" compared to "4340.277" in the table). This leads to an overall performance gain of HINDEX. In terms of disk reads, it is noteworthy that HINDEX incurs zero costs in the online phase.

TABLE II: Overhead under Put and Compact operations

Name	Exec. time (sec)		Number of disk reads	
HINDEX	**1553.158**		60699	
Update-in-place index	4619.456		313662	
Name	**Online**	**Offline**	**Online**	**Offline**
HINDEX	**1093.832**	459.326	0	60699
Update-in-place index	4340.277	**279.179**	252964	60698

VII. Related Work

Data indexing in scalable data management systems in cloud has been recently received many research attentions. HyperDex [12] is the first key-value store that supports a native ReadValue operation. The index in HyperDex is designed to support multi-dimensional attributes. It employs

a space-filling curve to reduce data dimensionality and shard data table on the reduced space. However, this approach can only scale to a moderate number of indexable attributes. To maintain the index, HyperDex treats it the same as a data replication process; The value-dependent chaining technique propagates the data updates to all replicas/indexes and essentially employs an update-in-place paradigm to relocate an updated object from the old place to a new one. Megastore [17] is Google's effort to support the cloud-scale database on the BigTable storage [1]. Megastore provides secondary index at two levels, namely the local index and the global index. The local index indexes the local data from a small entity group that is close to the local machine. The local index can be maintained synchronously at a fairly low cost. The global index which spans cross multiple groups is maintained in an asynchronous and lazy fashion. F1 [18], built on top of Spanner, supports global indexing in fully consistent and transactional way; It applies 2PC at a reasonable cost. PIQL [11] supports a scale-independent subset of SQL queries on the key-value stores which includes the value-based selection. The index management in PIQL uses the update-in-place paradigm; It deletes all the stale index entries upon an online update. PIQL is implemented as a library centric database purely on top of key-value store, while our HINDEX spans across both the client and server sides of a key-value store. To supports complex analytical queries, prior work [19] maintains a global materialized view asynchronously and a local materialized view synchronously on top of PNUTS [20]. In particular, the local view is mainly utilized for processing aggregations, the global view assists to evaluate selection queries based on secondary attributes. Secondary data index has been recently discussed in the open source community, such as for HBase [21] and Cassandra [22]. These key-value store indexes adopt the hidden table design to materialize the index data in key-value stores. In particular for HBase, it is proposed (though not released) to use append-only online indexing. However, it does not address any index repair process and may suffer from an eventually inconsistent index which further causes unnecessary cross-table checking during query processing. In addition, all existing work for scalable index/view support on key-value stores is not aware of the asymmetric performance in a write-optimized store and does not optimize the expensive index repair tasks, which the HINDEX design addresses.

VIII. Conclusion

This paper proposes HINDEX, a lightweight real-time indexing framework for generic log-structured key-value stores. The core design in HINDEX is to perform the *append-only* online indexing and compaction-triggered offline indexing. By this way, the online index update does not need to look into historic data for in-place updates, but rather appends a new version, which substantially facilitates the execution. To fix the obsolete index entries caused by the append-only indexing, HINDEX performs an offline batched index repair process. By coupling with the native compaction in a LSKV store, the batch index repair enjoys significant performance gain without incurring any extra disk overhead. We implemented a HINDEX prototype based on HBase and demonstrate the performance gain by a series of real-world experiment conducted in an Emulab cluster.

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, B. N. Bershad and J. C. Mogul, Eds. USENIX Association, 2006, pp. 205–218.

[2] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[3] "http://cassandra.apache.org/."

[4] "http://hbase.apache.org/."

[5] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[6] "http://code.google.com/p/leveldb/."

[7] "http://rocksdb.org."

[8] R. Sears and R. Ramakrishnan, "blsm: a general purpose log structured merge tree," in *SIGMOD Conference*, 2012, pp. 217–228.

[9] M. Yabandeh and D. G. Ferro, "A critique of snapshot isolation," in *EuroSys*, 2012, pp. 155–168.

[10] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 251–264.

[11] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. J. Franklin, and D. A. Patterson, "Piql: Success-tolerant query processing in the cloud," *PVLDB*, vol. 5, no. 3, pp. 181–192, 2011.

[12] R. Escriva, B. Wong, and E. G. Sirer, "Hyperdex: a distributed, searchable key-value store," in *SIGCOMM*, 2012, pp. 25–36.

[13] "http://www.emulab.net/."

[14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI*, D. E. Culler and P. Druschel, Eds. USENIX Association, 2002.

[15] "http://hadoop.apache.org/."

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[17] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011, pp. 223–234.

[18] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. O. K. Littlefield, D. Menestrina, S. E. J. Cieslewicz, I. Rae *et al.*, "F1: A distributed sql database that scales," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, 2013.

[19] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan, "Asynchronous view maintenance for vlsd databases," in *SIGMOD Conference*, 2009, pp. 179–192.

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "Pnuts: Yahoo!'s hosted data serving platform," *PVLDB*, vol. 1, no. 2, pp. 1277–1288, 2008.

[21] L. Hofhansl, "http://hadoop-hbase.blogspot.com/2012/10/musings-on-secondary-indexes.html."

[22] "http://hadoop-hbase.blogspot.com/2012/10/musings-on-secondary-indexes.html."

[23] "https://blogs.apache.org/hbase/entry/coprocessor_introduction."

[24] "https://issues.apache.org/jira/browse/cassandra-1016."

[25] "https://issues.apache.org/jira/browse/cassandra-1311."

Appendix A
Extension Interfaces in LSKV store

Following the trend of moving computation close to data, there is a recent body of work that enriches server-side functionality by adding extension interfaces to the key-value stores including HBase's CoProcessor [23], Cassandra's Plugins/Triggers [24], [25], Percolators' trigger functionality [10]. These extension interfaces typically expose event-based programming hooks and allow client applications to easily inject code (e.g., stored procedure) into store servers and associate that with their internal events. With the interfaces, one can extend the functionality of a key-value server without changing its internal code.